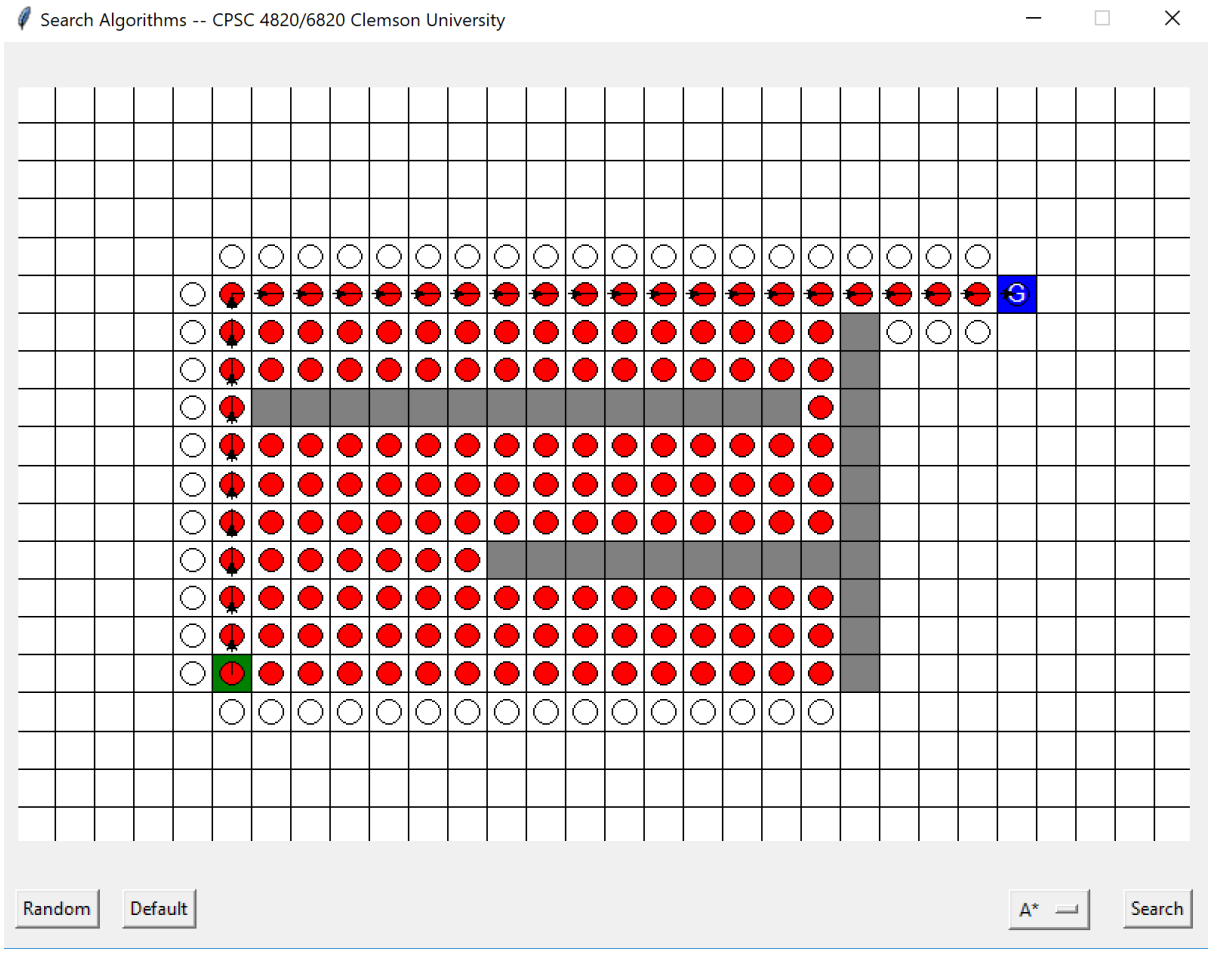

In the first programming assignment, you will implement a number of search algorithms to guide an agent through a grid-based world. For simplicity, we'll assume that the state space is 2-dimensional, consisting of the grid location of the agent.

Basic Requirements

The code for this project consists of two Python files. You can download them as a zip archive from here. After downloading the code, you should be able to display the grid world and generate different maps through the GUI by typing the following at the command line:

```
python search.py
```

Your task is to modify `search.py` in order to implement three of the search algorithms covered in class: depth-first search, uniform cost search, and A*. Note that as different algorithms differ only in the details of how the open list is managed, if you have the DFS working correctly, implementing the rest of the methods should be straightforward. Indeed, one possible implementation requires only a single generic search method that varies the queuing strategy. However, in this project, we'll keep the implementation of different methods separately.

Question 1 (2 points) Depth First Search

Modify the `depth_first_search` function in `search.py` to implement the depth-first search algorithm. You should employ the graph search version of DFS which allows a

state to be expanded only once. Test your code on different maps to evaluate its correctness. We've provided a number of default maps, but you're welcome to edit the `search_app.py` and create your own maps.

Question 2 (3 points) Uniform Cost Search

Depth-first search cannot find optimal paths, unless it gets lucky. To address this issue, modify the `uniform_cost_search` function in `search.py` to implement the uniform cost search algorithm. Again, write a graph search algorithm that allows a state to be expanded only once. Your algorithm should return the optimal path from start to goal.

Hint: As all action costs in the grid world are unit costs, uniform cost search behaves like breadth-first search with both methods returning optimal solutions (breaking ties in the priority queue may result in visually different minimal cost routes). You can verify this by implementing BFS by simply changing the data structure used in your DFS implementation to expand the open list nodes in a FIFO manner. Of course, when the step costs vary, BFS cannot guarantee optimal paths. You can change the cost function in `search_app.py` to verify this.

Question 3 (5 points) A* search

Modify the `astar_search` function in `search.py` to implement A* graph search. As A* needs a heuristic function to work, please modify the empty `heuristic` function accordingly to return the Manhattan distance between a given state and the goal state.

You should see that in general A* finds the optimal solution (slightly) faster than UCS. Note once more that the way you break ties in the priority queues may lead to different minimal cost paths for the two methods.

Hint: By setting the heuristic distance to 0, the A* search effectively behaves like uniform cost search. By comparing the two search algorithms, you should be able to see the advantages of heuristics search on certain scenarios.